min—max函数：

minmax函数是我们算法的核心之一，与价值算法一道，共同完成我们的决策。我们完成的minmax函数附带有α-β剪枝，用以避免多余的无意义的算力消耗，具体实现说明如下：

1. 结构方面：

（1）由于该决策需要众多参数，考虑诸多情况，为避免函数体过长导致调试困难和容错率低容易出bug的问题，在实现层面对该函数进行拆分，分为比较对手决策，取最小值的\_min函数和实现自己决策，取最大值的\_max函数。两个函数的深层搜索都是通过迭代，引用对方实现的。

（2）\_min和\_max函数的搜索深度为4，覆盖一个回合的四次决策，即每次决策都要考察双方的一次落子和一次合并。并用整数n来记录，初始值为3，每进行一层搜索就减少1，其值为0时停止，返回对应局势的价值函数。

2. 实现方面：

两个函数的基本功能类似，只在具体的选取最大值和最小值上有所区别。

（1）首先关于参数，函数需要的参数有：

· Board：棋盘类型，当前棋盘

· player布尔值，代表利用该函数决策的玩家，\_max函数player是我们自己，\_min函数的player是对手

· mode：字符串，方向和位置，要求作出的行动，是合并还是落子

· n：记录搜索深度，到达要求的深度就停止

· currentRound：代表当前回合数

· a，b：整数，记录棋面的价值，是α-β剪枝中的两个重要参数，初始值为-99999和99999

（2） 关于函数的本体：

· 终止条件：函数本身是一个相互引用的迭代函数，其终止条件是记录搜索深度的n变为0，如果为0，直接利用价值函数返回此时局面的价值。

· 规模的缩小：每一次决策都会将记录深度的n减1，并重新调用。

· 形势的判断：首先判断要求给出合并还是落子：

对于落子，分为三种情况：

如果己方区域有位置可以下，那就直接下在己方，然后进入下一层，在\_min中引用\_max作为下一层，\_max中引用\_min作为下一层。

如果己方无位置可下，则获取对方区域的空位，对每个空位都进行一次判断，\_max选择价值最大的落子，\_min则选择价值最小的落子。

如果没有地方可以落子，就直接到下一层。

需要注意的是，先手和后手有明显的区别，涉及到下一步是合并还是落子，因 此在进入下一层是需要单独判断。

对于合并，分为两种情况：

如果可以合并，就依次访问四个合并方向，判断哪个好，然后根据这一步做出决策的玩家选择最大值或者最小值，并进入下一层。

如果四个方向都不可以合并，那就直接进入下一层。

需要注意的是，先手和后手有明显的区别，涉及到下一步是合并还是落子，因此在进入下一层是需要单独判断。

· α-β剪枝：引入a作为\_max记录决策的最大值，b作为\_min记录决策的最小值，下一层会继承上一层的a，b值，对于\_min, 如果出现下一层\_max返回的值大于上一层的已有的最小值就直接枝剪，不进行进一步计算；对于\_max，如果出现下一层\_min返回的值小于已有的最大值，也直接剪枝，不再进行搜索。通过这样完成枝剪。